1. **Even Odd using 2 Channel & 1 Goroutine**

package main

import "fmt"

func main() {

evenCh := make(chan int)

oddCh := make(chan int)

go print(oddCh,evenCh)

for i:=0; i<=10; i++ {

if i%2==0 {

evenCh<-i

}else{

oddCh<-i

}

}

}

func print(oddCh, evenCh <-chan int){

for {

select {

case i:=<-oddCh :

fmt.Println(i)

case i:=<-evenCh :

fmt.Println(i)

}

}

}

1. **Even Odd using 2 Channel & 2 Goroutine**

package main

import (

"fmt"

"sync"

)

func main() {

evenCh := make(chan int)

oddCh := make(chan int)

var wg sync.WaitGroup

wg.Add(2)

go even(evenCh, &wg)

go odd(oddCh, &wg)

go func() {

wg.Wait()

close(evenCh)

close(oddCh)

}()

for i := 0; i < 6; i++ {

fmt.Println(<-evenCh)

fmt.Println(<-oddCh)

}

}

func even(evenCh chan int, wg \*sync.WaitGroup) {

defer wg.Done()

for i := 0; i <= 10; i = i + 2 {

evenCh <- i

}

}

func odd(oddCh chan int, wg \*sync.WaitGroup) {

defer wg.Done()

for i := 1; i <= 10; i = i + 2 {

oddCh <- i

}

}

1. **Even Odd using 1 Channel & 2 Goroutine**

package main

import (

"fmt"

"time"

)

func main() {

ch := make(chan bool)

go Even(ch)

ch <- true

go Odd(ch)

time.Sleep(time.Second \* 10)

}

func Odd(c chan bool) {

for i := 1; i < 10; {

<-c

fmt.Println("value is Odd ", i)

i += 2

c <- true

}

}

func Even(c chan bool) {

for i := 0; i < 10; {

<-c

fmt.Println("value is even ", i)

i += 2

c <- true

}

}

1. **Even Odd using 1 Channel & 2 Goroutine**

package main

import (

"fmt"

"time"

)

func main() {

ch := make(chan bool)

go Even(ch)

ch <- true

go Odd(ch)

time.Sleep(time.Second \* 10)

}

func Odd(c chan bool) {

for i := 1; i < 10; {

<-c

fmt.Println("value is Odd ", i)

i += 2

c <- true

}

}

func Even(c chan bool) {

for i := 0; i < 10; {

<-c

fmt.Println("value is even ", i)

i += 2

c <- true

}

}

1. Semaphore

func worker(id int, sem chan struct{}) {

defer wg.Done()

sem <- struct{}{}

fmt.Printf("Worker %d started\n", id)

fmt.Printf("Worker %d finished\n", id)

<-sem // Release semaphore (increase resource count)

}

func main() {

sem := make(chan struct{}, 3)

for i := 1; i <= 5; i++ {

wg.Add(1)

go worker(i, sem)

}

wg.Wait()

}

1. Worker Pool

func worker(id int, jobs <-chan int) {

defer wg.Done()

for j := range jobs {

fmt.Println("worker", id, "started job", j)

time.Sleep(time.Second)

fmt.Println("worker", id, "finished job", j)

}

}

func main() {

const numJobs = 5

jobs := make(chan int, numJobs)

for w := 1; w <= 3; w++ {

wg.Add(1)

go worker(w, jobs)

}

for j := 1; j <= numJobs; j++ {

jobs <- j

}

close(jobs)

wg.Wait()

}

**7 ) Batch**

package main

import (

"fmt"

"sync"

)

func main() {

data := []int{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

batch := 5

var wg sync.WaitGroup

// Create a buffered channel to store the sums of each batch

ch := make(chan int, len(data)/batch+1)

for i := 0; i < len(data); i += batch {

end := i + batch

if end > len(data) {

end = len(data)

}

wg.Add(1)

go add(data[i:end], ch, &wg)

}

// Wait for all goroutines to complete

wg.Wait()

close(ch)

// Collect and sum up the results from the channel

totalSum := 0

for sum := range ch {

totalSum += sum

}

fmt.Println("Total : “ totalSum)

}

func add(arr []int, ch chan int, wg \*sync.WaitGroup) {

defer wg.Done()

sum := 0

for \_, n := range arr {

sum += n

}

ch <- sum

}

1. **Increment Atomic Package**

func main() {

var counter int64

var wg sync.WaitGroup

for i := 0; i < 100; i++ {

wg.Add(1)

go func() {

defer wg.Done()

atomic.AddInt64(&counter, 1)

}()

}

wg.Wait()

fmt.Println(counter)

}

1. **Select**

func main() {

ch1 := make(chan []int)

ch2 := make(chan []int)

ch3 := make(chan []int)

go imei1(ch1)

go imei2(ch2)

go imei3(ch3)

t := time.After(4 \* time.Second)

for {

select {

case i := <-ch1:

fmt.Println("IMEI 1 :", i)

case i := <-ch2:

fmt.Println("IMEI 2 :", i)

case i := <-ch3:

fmt.Println("IMEI 3 :", i)

case <-t:

fmt.Println("TIMEOUT")

return

}

}

}

func imei1(ch chan []int) {

time.Sleep(1 \* time.Second)

s := []int{1, 2, 3, 4}

ch <- s

}

func imei2(ch chan []int) {

time.Sleep(2 \* time.Second)

s := []int{5, 6, 7, 8}

ch <- s

}

func imei3(ch chan []int) {

time.Sleep(5 \* time.Second)

s := []int{9, 10, 11, 12}

ch <- s

}

1. **Producer & Consumer**

package main

import (

"fmt"

"math/rand"

"sync"

)

func main() {

ch := make(chan int, 10)

var producerWg sync.WaitGroup

var consumerWg sync.WaitGroup

for i := 1; i <= 3; i++ {

producerWg.Add(1)

go producer(i, ch, &producerWg)

}

// Start 2 consumers

for i := 1; i <= 2; i++ {

consumerWg.Add(1)

go consumer(i, ch, &consumerWg)

}

producerWg.Wait()

close(ch)

consumerWg.Wait()

}

func producer(id int, ch chan int, wg \*sync.WaitGroup) {

defer wg.Done()

for i := 0; i < 2; i++ {

r := rand.Intn(100)

fmt.Printf("Producer %d produced: %d\n", id, r)

ch <- r

}

}

func consumer(id int, ch chan int, wg \*sync.WaitGroup) {

defer wg.Done()

for n := range ch {

fmt.Printf("Consumer %d consumed: %d\n", id, n)

}

}

1. **Producer-Consumer M\*N**

package main

import (

"fmt"

"sync"

"sync/atomic"

)

var sum int

var sumMutex sync.Mutex

var index int32

func main() {

ch := make(chan int)

var producerWg sync.WaitGroup

var consumerWg sync.WaitGroup

data := []int{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

m ,n := 3,2

for i := 1; i <= m; i++ {

producerWg.Add(1)

go producer(i, data, ch, &producerWg)

}

for i := 1; i <= n; i++ {

consumerWg.Add(1)

go consumer(i, ch, &consumerWg)

}

producerWg.Wait()

close(ch)

consumerWg.Wait()

fmt.Printf("Final sum of consumed values: %d\n", sum)

}

func producer(id int, data []int, ch chan int, wg \*sync.WaitGroup) {

defer wg.Done()

for {

i := int(atomic.AddInt32(&index, 1)) - 1

if i >= len(data) {

break

}

value := data[i]

fmt.Printf("Producer %d produced: %d\n", id, value)

ch <- value

}

}

func consumer(id int, ch chan int, wg \*sync.WaitGroup) {

defer wg.Done()

localSum := 0

for value := range ch {

fmt.Printf("Consumer %d consumed: %d\n", id, value)

localSum += value

}

sumMutex.Lock()

sum += localSum

sumMutex.Unlock()

}

1. **Even Odd**

func main() {

evenCh := make(chan int)

oddCh := make(chan int)

var wg sync.WaitGroup

wg.Add(2)

go even(evenCh, oddCh, &wg)

go odd(evenCh, oddCh, &wg)

wg.Wait()

}

func even(evenCh, oddCh chan int, wg \*sync.WaitGroup) {

defer wg.Done()

for i:=2; i<=10; i+=2 {

fmt.Println("Odd :", <-oddCh)

evenCh<-i

}

close(evenCh)

}

func odd(evenCh, oddCh chan int, wg \*sync.WaitGroup) {

defer wg.Done()

for i:=1; i<=10; i+=2 {

oddCh<-i

fmt.Println("Even :", <-evenCh)

}

close(oddCh)

}

1. **Global Sum Buffered**

package main

import (

"fmt"

"sync"

)

var sum int

var mu sync.Mutex // To protect the global sum variable

func main() {

data := []int{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

readerCount := 5

writerCount := 3

ch := make(chan int, len(data)) // Buffered channel to hold the entire array

var wgReaders, wgWriters sync.WaitGroup

// Start reader goroutines

chunkSize := len(data) / readerCount

for i := 0; i < readerCount; i++ {

start := i \* chunkSize

end := start + chunkSize

if i == readerCount-1 { // Handle remaining data in the last chunk

end = len(data)

}

wgReaders.Add(1)

go reader(ch, data[start:end], &wgReaders)

}

// Start writer goroutines

for i := 0; i < writerCount; i++ {

wgWriters.Add(1)

go writer(ch, &wgWriters)

}

// Wait for all readers to finish

wgReaders.Wait()

close(ch) // Close the channel after all readers are done

// Wait for all writers to finish processing

wgWriters.Wait()

fmt.Println("Total sum:", sum)

}

func reader(ch chan int, data []int, wg \*sync.WaitGroup) {

defer wg.Done()

for \_, v := range data {

ch <- v // Send each element to the channel

}

}

func writer(ch chan int, wg \*sync.WaitGroup) {

defer wg.Done()

for v := range ch {

mu.Lock()

sum += v // Add each received value to the global sum

mu.Unlock()

}

}

1. **Global Sum Unbuffered**

package main

import (

"fmt"

"sync"

)

var sum int

var mu sync.Mutex // To protect the global sum variable

func main() {

data := []int{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

readerCount := 5

writerCount := 3

ch := make(chan int) // Unbuffered channel

var wgReaders, wgWriters sync.WaitGroup

// Start reader goroutines

chunkSize := len(data) / readerCount

for i := 0; i < readerCount; i++ {

start := i \* chunkSize

end := start + chunkSize

if i == readerCount-1 { // Handle remaining data in the last chunk

end = len(data)

}

wgReaders.Add(1)

go reader(ch, data[start:end], &wgReaders)

}

// Start writer goroutines

for i := 0; i < writerCount; i++ {

wgWriters.Add(1)

go writer(ch, &wgWriters)

}

// Wait for all readers to finish

wgReaders.Wait()

close(ch) // Close the channel after all readers are done

// Wait for all writers to finish processing

wgWriters.Wait()

fmt.Println("Total sum:", sum)

}

func reader(ch chan int, data []int, wg \*sync.WaitGroup) {

defer wg.Done()

for \_, v := range data {

ch <- v // Send each element to the channel

}

}

func writer(ch chan int, wg \*sync.WaitGroup) {

defer wg.Done()

for v := range ch {

mu.Lock()

sum += v // Add each received value to the global sum

mu.Unlock()

}

}